Making Sense of Actor Behaviour: An Algebraic Filmstrip Pattern and its Implementation

Tony Clark
Aston University
Birmingham, UK
tony.clark@aston.ac.uk

Balbir Barn
Middlesex University
London, UK
b.barn@mdx.ac.uk

Vinay Kulkarni, Souvik Barat
TCS Research
Pune, India
vinay.vkulkarni@tcs.com
souvik.barat@tcs.com

ABSTRACT

Sense-making with respect to actor-based systems is challenging because of the non-determinism arising from concurrent behaviour. One strategy is to produce a trace of event histories that can be processed post-execution. Given a semantic domain, the histories can be translated into visual representations of the semantics in the form of *filmstrips*. This paper proposes a general pattern for the production of filmstrips from actor histories that can be implemented in a way that is independent of the particular data types used to represent the events, semantics and graphical displays. We demonstrate the pattern with respect to a simulation involving predators and prey which is a typical agent-based application.

CCS CONCEPTS

• **Software and its engineering → Concurrent programming structures**; **Software testing and debugging**.

KEYWORDS

Actors, Filmstrips

ACM Reference Format:
Tony Clark, Balbir Barn, and Vinay Kulkarni, Souvik Barat. 2019. Making Sense of Actor Behaviour: An Algebraic Filmstrip Pattern and its Implementation. In *12th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference) (ISEC'19), February 14–16, 2019, Pune, India.* 10 pages. https://doi.org/10.1145/3299771.3299783

1 INTRODUCTION

Systems such as smart energy-grids, supply-chain networks and smart factories can be represented using Multi-Agent Systems (MAS) [15, 27, 35] where systems are constructed in terms of independent goal-directed agents that concurrently engage in tasks both independently and collaboratively. The benefits of MAS include resilience [14] and adaptation [3] which are desirable properties for modern complex distributed heterogeneous systems. MAS can also be used to develop simulations of systems [16]. An important reason for using agents for simulation is that the systems of interest are complex and involve, for example, socio-technical features [26].

ISEC'19, February 14–16, 2019, Pune, India

ACM ISBN 978-1-4503-6215-3/19/02…$15.00
https://doi.org/10.1145/3299771.3299783

MAS are inherently non-deterministic and exhibit emergent behaviour which makes debugging and sense-making challenging [33, 40]. Recent work on MAS verification has focussed on static analysis of the communication between agents [34] using interaction protocols [1].

Sense-making incorporates a range of tasks. Such tasks consist of information gathering, re-representation of the information in a schema that aids analysis, the development of insight through the manipulation of this representation, and the creation of some knowledge product or direct action based on the insight [32]. A specialised form of sense-making is debugging. Sense-making can be supported through the use of domain-specific representations of system execution [33]. Augmenting the temporal aspect by a visual representation of the execution data gives improved understanding of systems [2].

Our hypothesis to address these challenges is the use of histories and their subsequent manipulation to perform sense-making. Each agent can produce a history that consists of a description of its local state changes. However, the resulting collection of histories requires combination in the context of a semantic model in order to meaningfully represent the history of a complete system. Therefore, our proposal imposes a semantics on the histories in order to support sense-making. Furthermore, since it is a history, we would like to be able to 'play' the history forwards and backwards to understand what happened during the execution traces over time.

The contribution of this paper is a semantics-based filmstrip pattern that can be used to support MAS sense-making that is independent of any particular implementation technology. Filmstrips are generally attributed to D'Souza and Wills in their modelling method, Catalysis [12]. A filmstrip is a sequence of snapshots (objects and relationships) describing system state transitions arising from operation calls in the system. The proposal is evaluated in terms of an implementation using the actor-based language ESL.

This paper motivates the use of filmstrips as a basis for analyzing agent-based systems in section 2 using a standard MAS application involving predators and prey. This is a typical agent-based application [13] that is applied to understanding community dynamics [11], ecology [38], and infectious diseases [37]. The application is written in an actor language ESL [9] whose semantics is presented in section 2.2.

The main contribution of the paper is given as an algebraic pattern in section 3 that can be used to construct filmstrips independently of the data types that are used to represent the event histories and the semantics of the application. The pattern is then

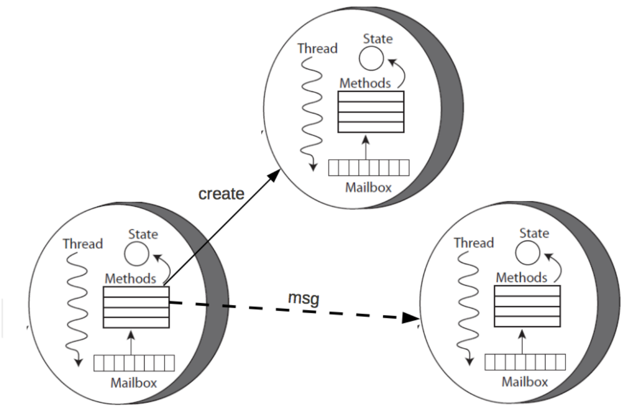


Figure 1: Actor Model of Computation [21]

implemented in section 4 using ESL polymorphic functions to abstract from the implementation data types. The pattern is used to build predator-prey filmstrips in section 5.

The paper concludes with an overview in section 6 of several filmstrip implementations using the ESL-based pattern and reviews related work in section 7.

2 ACTORS AND FILMSTRIP GENERATION

Figure 1 (taken from [21]) shows the key features of the actor-model of computation. Each actor is associated with a single thread of control, some state, a mailbox queue and some message handling methods. Messages are sent between actors asynchronously and added to the receiver's mailbox. When an actor is idle, the next message in the mailbox is inspected and handled to the appropriate method whose body is performed on the thread.

Since each actor is autonomous, groups of actors are often used to create simulations of populations in order to observe their behaviour. A typical example of this is the *predator-prey* simulation [30] where a group of predators (in this case wolves) try to catch a prey (in this case a sheep). The purpose of the simulation is to investigate different strategies employed by each category of actor: predators try to catch the prey who in turn tries to evade them. Section 2.1 describes an implementation of predator-prey in the language ESL which is defined in section 2.2. The issues arising from the use of filmstrips for debugging is described in section 2.3 leading to the definition of sense-making requirements to be addressed by the pattern defined in the following section.

2.1 Predator Prey Filmstrips

Figure 2 shows an ESL program that implements a simple version of predator-prey. This section gives an informal description of the program and section 2.2 provides a formal definition that is necessary to precisely capture the debugging challenge and subsequent definition of the filmstrip pattern.

ESL combines functional and actor-based programming [8–10] making it an ideal candidate for the proposed filmstrip pattern. An ESL program consists of a collection of value, function and behaviour definitions. Each behaviour has a corresponding type

```
1  type Predator = Act { Move }
2  type Prey     = Act { Move }
3  type Main     = Act { Time(Int) }
4  data Message  = PredAt(Int,Int,Int) | PreyAt(Int,Int);
5  data Pos      = Point(Int,Int);
6
7  messages ∷ [Message] = [];
8
9  act predator(id∷Int,x∷Int,y∷Int)∷Predator {
10   Move → grab(messages) {
11     let dx∷Int = randomMove();
12         dy∷Int = randomMove()
13     in if isNearerPrey(id,dx,dy) and canMove(x+dx,y+dy)
14        then {
15          x := x + dx;
16          y := y + dy;
17          messages := messages + [PredAt(x+dx,y+dy)];
18        }
19   }
20 }
21
22 act prey(x∷Int,y∷Int)∷Prey {
23   Move → grab(messages) {
24     let dx∷Int = randomMove();
25         dy∷Int = randomMove()
26     in if isAwayFromPred(dx,dy) and canMove(x+dx,y+dy)
27        then {
28          x := x + dx;
29          y := y + dy;
30          messages := messages + [PreyAt(x+dx,y+dy)];
31        }
32   }
33 }
34
35 predators∷[Predator] =
36   [ new predator(p,random(width),random(height))
37   | p∷Int ← 0..numOfPredators ];
38
39 thePrey∷Prey = new prey(random(width),random(height));
40
41 rocks∷[Pos] =
42   [ Point(random(width),random(height))
43   | r∷Int ← 0..numOfRocks ];
44
45 act main∷Main {
46   Time(n∷Int) when n < limit → {
47     for p∷Predator in predators do
48       p ← Move;
49     thePrey ← Move;
50     wait(1);
51   }
52   Time(n∷Int) → {
53    showFilmstrip(messages);
54    stopAll();
55   }
56 }
```

Figure 2: ESL Definition Predator-Prey Behaviours

definition listing the messages that can be received by any actor with the behaviour. Example behaviour types are listed in lines 1–3; `Predator` and `Prey` both define a message `Move`, and `Main` defines a `Time` message. The latter is key to the ESL semantics which provides all actors with a message telling them the current time at regular intervals and which can be used to drive the application and eventually terminate it (line 54).

Line 4 defines a union data type `Message` that has two alternatives: `PredAt(i,x,y)` meaning that a predator with identity `i` is at position `(x,y)`, and `PreyAt(x,y)` meaning that the prey is at position `(x,y)`. The data value `messages` on line 7 is a list of messages and is initialised to the empty list. This will be used as the application history with all actors posting messages to the end of the list.

The behaviours `predator` (lines 9–20) and `prey` (lines 22 – 33) define what happens when an actor with the respective behaviours handles a `Move` message. Both behaviours have initialisation arguments that

grab(v)	$$\frac{v \notin \gamma}{[E;\gamma \vdash \mathbf{grab}(v,e)]_a \rightarrow_\lambda [E;\gamma,v \vdash e,\mathbf{release}(v)]_a}$$
release(v)	$[E;\gamma,v \vdash \mathbf{release}(v)]_a \rightarrow_\lambda [E;\gamma \vdash nil]_a$
fun(a)	$$\frac{[E;\gamma \vdash e]_a \rightarrow_\lambda [E';\gamma' \vdash e']_a}{\langle \alpha,[E \vdash e]_a \mid \mu;\gamma\rangle \rightarrow \langle \alpha,[E' \vdash e']_a \mid \mu;\gamma'\rangle}$$
new(a,a',E',b')	$\langle \alpha,[E \vdash R[\mathbf{new}(a',b',E')]]_a \mid \mu;\gamma\rangle \rightarrow$ $\langle \alpha,[E \vdash R[nil]]_a,(E,E' \vdash b')_{a'} \mid \mu;\gamma\rangle$
term(a,b)	$\langle \alpha,[E \vdash R[]]_a \mid \mu;\gamma\rangle \rightarrow \langle \alpha,(E \vdash b)_a \mid \mu;\gamma\rangle$
rcv(a,v)	$\langle \alpha,(E \vdash b)_a \mid \mu,(a \Leftarrow v);\gamma\rangle \rightarrow$ $\langle \alpha,[E(FV(b) \mapsto v) \vdash b[nil]]_a \mid \mu;\gamma\rangle$
snd(a,a',v)	$\langle \alpha,[E \vdash R[\mathbf{send}(a',v)]]_a \mid \mu,\gamma\rangle \rightarrow$ $\langle \alpha,[E \vdash R[nil]]_a \mid \mu,(a' \Leftarrow v);\gamma\rangle$
time(t)	$\langle \alpha \mid \mu;\gamma\rangle \rightarrow \langle \alpha \mid \mu,\{(a \Leftarrow \mathrm{Time}(t)) \mid a \in \alpha\};\gamma\rangle$

Figure 3: ESL Operational Semantics

are used as an actor's state. ESL implements lexical scoping so that variables are local within the text contained within their defining occurrence. In addition, variables can be changed by side-effect, therefore, the variables `x` and `y` at line 9 are both private to the `predator` behaviour and form the mutable state of any actor with that behaviour.

Both behaviours handle `Move` similarly. They grab the history (lines 10 and 23) providing the actor with exclusive access. Both behaviours define the movement strategy in terms of some functions that are omitted: predators try to catch the prey and the prey aims to avoid the predators. In both cases if the receiver decides to move, the local state is updated and a message is added to the end of the global list `messages`.

A list of predators is created in lines 35–37 and a single prey is created in line 39. A list of rock positions is created (lines 41–43); the details of keeping predator, prey and rock positions separate is omitted.

An ESL program starts by creating a single actor with the behaviour `main` defined on lines 45–56. The `Time` messages drive the main actor to send `Move` messages to each of the predator and prey actors. The messages are sent asynchronously and Actor model of computation guarantees that the messages will be received and computation will be fairly distributed. Once the time limit is reached (lines 52–55) the application shows a filmstrip constructed from the history and stops the application. The rest of the paper describes how the filmstrip is constructed and displayed.

2.2 ESL

Actor-based systems are highly concurrent which makes debugging them a challenge. This section defines the semantics of ESL based on a standard actor semantics [29, 31] which is extended with monitors as used by **grab** in the previous section. The filmstrip pattern defined in ESL makes use of polymorphism in order to be independent of the semantic domain used as a basis for sense-making. This section also defines a type relation for polymorphic ESL that is suitable for the filmstrip pattern definitions.

Figure 3 defines the operational semantics of ESL. An ESL configuration is $\langle \alpha \mid \mu;\gamma\rangle$ where α is a set of actors, μ is a multi-set of pending messages and γ is a set of monitors that are currently locked. An actor $a \in \alpha$ can either be *busy* or *inactive*. A busy actor is represented as $[E \vdash R[e]]_a$ where E is the local state of the actor, and R is a reduction context filled with expression e that is currently being executed. An inactive actor is waiting for a message and is represented as $(E \vdash b)_a$ where b is its behaviour. A message to a that is pending is represented as $a \Leftarrow v$.

The language of ESL actor behaviours is standard (as noted in [31] and represented by **fun**(a)), the reduction relation $\rightarrow_\lambda$ in figure 3 is therefore not fully defined except for the novel feature of monitors given by rules **grab**(v), **release**(v) where the monitor v is added to, and removed from, the global set γ. Since the reduction relation $\rightarrow_\lambda$ is a single-step semantics, adding a monitor to γ provides exclusive access and causes other actors that concurrently attempt to grab the same monitor to wait until the monitor is released.

Rule **new**(a',E',b') differs from that given in [31] to note that a new actor captures both the current context E, but also creates its own local context E'. Since ESL supports side effects, this allows actors to share state that can be managed via monitors.

Rule **term**(a,b) applies when an actor exhausts its current message handler and becomes inactive. Rule **rcv**(a,v) shows how an inactive actor starts to process a message and rule **snd**(a,a',v) describes message passing.

Rule **time**(t) injects `Time`(t) messages into the actor community. ESL does not define when these messages occur - they are used to ensure that otherwise idle actors can regularly perform computation and are provided with time t in milliseconds since the start of the application.

The semantic relation $\rightarrow$ defined in figure 3 places no further constraints on the order in which actor execution proceeds. Behaviour is highly concurrent and message passing is asynchronous making it difficult to trace threads of execution.

ESL is a statically typed language that merges features from functional programming and actor-languages. The filmstrip pattern that is defined in ESL in section 4 is independent of the data type used to represent the semantic domain used to structure the snapshots. Figure 4 defines that part of ESL type relation used by the examples in this paper. The relation is defined as $\Gamma \vdash$ `t::T` where Γ is a set of type associations for identifiers $x :: T$, `t` is a program term and `T` is a type.

Of particular interest is the ESL support for universal types and type application. An identifier can be defined to range over one or more types, for example:

```
pair[T](x::T)::[T] = [x,x]
```

that defines a function of type `∀T.(T)->[T]`. When the function is used, the type must be supplied `pair[Int](10)::[Int]`.

A data definition (as shown in figure 2 on line 4 introduces a union type. Such a type defines a number of constructors; in this case `PredAt` and `PreyAt` which are used to inject values into the union data type. Therefore, `PredAt(1,20,30)` is a value of type `Message`. Values

Syntax:

t ::=		*terms*
	x	*variable*
	new(t)	*creation*
	$\lambda(\bar{d})$ t	*function*
	act { $\bar{d}$ $\bar{h}$ }	*behaviour*
	n	*number*
	s	*string*
	b	*boolean*
	grab(x) t	*lock*
	if t **then** t **else** t	*conditional*
	t ← t	*send*
	case t { $\bar{h}$ }	*projection*
	[$\bar{t}$]	*list*
	t($\bar{t}$)	*application*
	t[$\bar{T}$]	*type application*
	C($\bar{t}$)	*injection*
i ::=	d=t	*initialisation*
d ::=	x[$\bar{X}$]::T	*declarations*
h ::=	C($\bar{x}$) → t	*handlers*
T ::=		*types*
	Int	*integer type*
	Bool	*boolean type*
	Str	*string type*
	∀ $\bar{X}$. T	*universal*
	X	*variable*
	Union { $\bar{m}$ }	*union*
	Act { $\bar{d}$ $\bar{m}$ }	*behaviour*
	($\bar{T}$) → T	*function type*
m ::=	C($\bar{T}$)	*message type*

Type Checking:

$$\frac{x :: T \in \Gamma}{\Gamma \vdash x :: T} \quad \text{T-VAR}$$

$$\frac{\Gamma \vdash t :: \textbf{Act}\ \{\ \bar{d}\ \bar{m}\ \}}{\Gamma \vdash \textbf{new}(t) :: \textbf{Act}\ \{\ \bar{d}\ \bar{m}\ \}} \quad \text{T-NEW}$$

$$\frac{\Gamma, \bar{x}::\bar{T} \vdash t::T}{\Gamma \vdash \lambda(\bar{d})\ t::(\bar{T})\rightarrow T} \quad \text{T-FUN}$$

$$\Gamma \vdash \textbf{act}\ \{\ \bar{d}\ \bar{m}\ \}::\textbf{Act}\ \{\ \bar{d}\ \bar{m}\ \} \quad \text{T-ACT}$$

$$\frac{\Gamma \vdash t_1::\textbf{Act}\,\{\,\bar{d}\ \bar{m}\,\} \qquad \Gamma \vdash t_2::\textbf{Union}\,\{\,\bar{m}\,\}}{\Gamma \vdash t_1 \leftarrow t_2::\textbf{Union}\,\{\,\bar{m}\,\}} \quad \text{T-SEND}$$

$$\Gamma \vdash n::\textbf{Int} \quad \text{T-INT}$$

$$\Gamma \vdash s::\textbf{Str} \quad \text{T-STR}$$

$$\Gamma \vdash b::\textbf{Int} \quad \text{T-BOOL}$$

$$\frac{\Gamma \vdash t::T}{\Gamma \vdash \textbf{grab}(x)\ t\ ::\ T} \quad \text{T-LOCK}$$

$$\frac{\Gamma \vdash t_1::\mathtt{Bool} \qquad \Gamma \vdash t_2, t_3::T}{\Gamma \vdash \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3\ ::\ T} \quad \text{T-IF}$$

$$\frac{\Gamma \vdash \textbf{case}\ t\ \{\bar{m}_1\} :: T \qquad \Gamma \vdash \textbf{case}\ t\ \{\bar{m}_2\} :: T}{\Gamma \vdash \textbf{case}\ t\ \{\bar{m}_1, \bar{m}_2\} :: T} \quad \text{T-CASE1}$$

$$\frac{\Gamma \vdash t_1 :: \textbf{Union}\{\bar{m}, C(\bar{T})\} \qquad \Gamma, \bar{x} :: \bar{T} \vdash t_2 :: T}{\Gamma \vdash \textbf{case}\ t_1\ \{C(\bar{x}) \rightarrow t_2\} :: T} \quad \text{T-CASE2}$$

$$\Gamma \vdash [] :: \forall X.[X] \quad \text{T-NIL}$$

$$\frac{\Gamma \vdash [\bar{t}_1] :: [T] \qquad \Gamma \vdash [\bar{t}_2] :: [T]}{\Gamma \vdash [\bar{t}_1, \bar{t}_2] :: [T]} \quad \text{T-LIST1}$$

$$\frac{\Gamma \vdash t :: T}{\Gamma \vdash [t] :: [T]} \quad \text{T-LIST2}$$

$$\frac{\Gamma \vdash \bar{t} :: \bar{T} \qquad \Gamma \vdash t :: (\bar{T}) \rightarrow T}{\Gamma \vdash t(\bar{t}) :: T} \quad \text{T-APP}$$

$$\frac{\Gamma \vdash t :: \forall \bar{X}.T}{\Gamma \vdash t[\bar{T}] :: T[\bar{T}/\bar{X}]} \quad \text{T-TAPP}$$

$$\frac{\Gamma, C(\bar{T}) \mapsto \textbf{Union}\{\bar{m}\} \qquad \Gamma \vdash \bar{t} :: \bar{T}}{\Gamma \vdash C(\bar{t}) :: \textbf{Union}\{\bar{m}\}} \quad \text{T-INJ}$$

Figure 4: ESL Type Checking

of a union type can be projected onto their constituent elements using a **case**-expression. Values of a union type are also used as messages in ESL where the message handlers are used to project the values.

2.3 Sense-Making Requirements

Our aim is to determine whether or not the ESL program defined in figure 2 exhibits the behaviour we expect. In general, it is difficult to achieve this through instrumentation due to the highly concurrent and non-deterministic nature of actor computation. In principle we could apply static verification techniques to the program to investigate a required behaviour, however for applications of any size the state-space explosion makes this approach unusable. Therefore, we propose to use a post-execution, human-based, machine-assisted

(a) Predators Move In **(b) Simple Prey Strategy**

Figure 5: Filmstrip

technique where the history of execution is analysed. Consider a system history for the predator-prey example:

```
PreyAt(10,20)          // The prey starts at (10,20)
PredAt(1,20,10)        // Predator 1 starts at (20,10)
PreyAt(9,20)           // The prey moves to (9,20)
PredAt(1,19,11)        // Predator 1 moves to (19,11)
```

Such a sequence of actions is difficult to interpret because the semantics of any global state is the aggregation of previous actions. Furthermore, some actions overwrite previous actions: the movement of predator 1 above. In order to make sense of any given history we propose a filmstrip that can be run forwards and backwards.

A filmstrip is a visual semantic description of the system in terms that allow us to spot issues of interest and to perform some sense-making analysis. A typical example of a filmstrip is shown in figure 5 where the sequence of predator-prey messages has been transformed into a sequence of *snapshots* displayed via a slider that can be dragged forwards and backwards to display different points in time. Figure 5a shows the predators moving towards the prey and figure 5b clearly shows the prey strategy to be unintelligent since the move places the sheep in a position that is surrounded by rocks on three sides.

The use of system visualisation and filmstrips in particular is a known technique for parallel systems [4, 24] and for MAS [39]. Whilst these approaches acknowledge the need to integrate events, semantics and displays, none provide a structure for doing so in the context of MAS. The following features are required to create such an integrated structure:

$$\begin{tikzcd}
([E], i : E \to \mathrm{Int}) \arrow[r, "\sigma"] \arrow[d, "\omega"'] & ([[E]], i : E \to \mathrm{Int}) \arrow[d, "\mu^*"] \\
{[(D, \otimes, \delta)]} & {[(S, \oplus, \epsilon)]} \arrow[l, "\gamma^*"]
\end{tikzcd}$$

Figure 6: The Filmstrip Pattern

Event History The application must produce a history of events in a form that can be aggregated to produce snapshots as described below.

Semantic Domain The history consists of individual messages that must be aggregated based on a semantics for the application. It is useful if the semantic domain is compositional since the individual components of the history are produced by different actors.

State Transitions The history of the application must be mapped to a sequence of state transitions each of which is defined as a semantic snapshot.

Display Domain Each snapshot must be mapped to a visual representation such as that described in figure 5. The displays should be designed so as to exhibit behaviour of interest.

3 THE FILMSTRIP PATTERN

This paper proposes a filmstrip pattern that is independent of the semantic and display domains that are used. The pattern places conditions on these domains but leaves the details to the particular applications. This section defines the pattern; section 3.1 provides some basic definitions, section 3.2 presents the pattern definition, section 3.3 describes how messages are translated to state transitions, section 3.4 describes how the transitions are translated to semantic values, section 3.5 describes how semantic values are translated to displays. The pattern is independent of particular data types, however section 3.6 states the properties for any data types that are used.

3.1 Basic Definitions

A *monoid* $(M, m_0, +)$ is a set of values M together with a value $m_0 : M$ and an associative binary operation $\_ + \_ : (M, M) \to M$ such that m_0 is the left and right identity of $+$. Given a list of values $xs : [X]$ the lists $\uparrow xs$ and $\downarrow xs$ are defined to be the prefix of xs and the last element of xs respectively such that $\uparrow xs + [\downarrow xs] = xs$. The function $[\downarrow]$ is defined $[\downarrow](xs) = [\downarrow xs]$. Given a function $f : X \to Y$, the function $f^* : [X] \to [Y]$ maps f over a list of type $[X]$ to produce a list of type $[Y]$. Given an associative binary operator $\_ * \_ : (Y, Y) \to Y$, and a value $y : Y$ the function $\backslash_{f,*,y}$ maps a list $[x_1, x_2, ...]$ to produce $f(x_1) * f(x_2) * ... * y$. $|l|$ is the length of the list l.

3.2 Pattern Definition

An actor-based system executes in terms of messages. When a message is received by an actor it may change state. The state changes can be recorded as events which, over the duration of an application, build up a history of execution. Each actor has a unique identity which can be used to tag the events it produces leading to a structure $([E], i : E \to \mathrm{Int})$ of event histories.

A filmstrip $f : [D]$ is a sequence of display elements. The displays represent elements that can be drawn on a screen and have no knowledge of system executions. We would like to define a mapping $\omega : [E] \to [D]$ from sequences of events to filmstrips that preserves a semantic structure that we define for the system. The data type for displays should be defined so that it forms a monoid $(D, \otimes, \delta)$ where $\delta : D$ is the empty display and $\_ \otimes \_ : (D, D) \to D$ composes displays.

The mapping $\omega : [E] \to [D]$ from histories to filmstrips is to be defined in terms of three mappings: $\sigma : [E] \to [[E]]$ that maps event histories to state transitions; $\mu^* : [[E]] \to [S]$ that maps sequences of state transitions to sequences of semantic values; $\gamma^* : [S] \to [D]$ that maps sequences of semantic values to sequences of displays.

The pattern is defined in figure 6; each of the components are defined in the rest of this section.

3.3 Producing State Transitions

A system state can be expressed as a collection of facts that describe the current state of each actor. If the event history contains a record of the complete state of an actor each time it changes then a sequence of events can be transformed into a sequence of states by taking all the prefixes of the history. However, states produced in this way may contain contradictory facts about a given actor since the state may change over time. Therefore, we must filter the prefixes so that the latest state of each actor is retained. The mapping σ is defined by specifying its inverse $\sigma^{-1} :: [[E]] \to [E]$: $\sigma^{-1}(ess) = es$ such that the following two conditions hold:

$$\backslash_{[\downarrow],+,[]}(ess) = es \tag{1}$$

$$\forall_{j \in 1..\#(ess)} \uparrow ess_j = [m \mid m \in ess_{j-1}, i(m) \neq id(\downarrow ess_j)] \tag{2}$$

$$\forall_{j \in 1..\#(ess)} |ess_j| = |ess_{j-1}| + 1 \tag{3}$$

Condition 1 states that the concatenation of the last element of each state must produce the original history. Condition 2 states that the prefix of each state must not contain a message whose id is that of the suffix. Together, these conditions ensure that σ generates a step-by-step state transition that does not contain contradictory information about any element. Condition 3 requires the state transitions to be incremental.

3.4 Producing Semantic Values

A key feature of the pattern is the requirement to define a semantic domain S that is used as the anchor-point of filmstrip production. The semantics is defined in order to reflect the features of the domain that we would like to examine. For example in the case of the predator-prey scenario, the semantic domain is a world containing positions of the predators and prey. Other semantic domains may be more complex, however there is a requirement that the domain can be expressed as a monoid in order that it has an empty element and a composition operator. This allow the mapping between sequences of system states $[[E]]$ and sequences of semantic values $[S]$ to be defined in terms of a simple mapping $e : E \to S$ between events and semantic values such that: $\mu = \backslash_{e,\oplus,\epsilon}$ and therefore $\mu^* : [[E]] \to [S]$ as required.

3.5 Producing Displays

Given that we have defined filmstrips as a monoid over displays it is possible to define the mapping γ^* in terms of a simple display mapping $d : S \rightarrow D$ since this can be generalised in the same way as e above:

$$\begin{array}{lcl} \gamma([]) & = & [] \\ \gamma(\epsilon) & = & \delta \\ \gamma(s) & = & d(s) \\ \gamma(s_1 \oplus s_2) & = & \gamma(s1) \otimes g(s_2) \end{array}$$

The individual mappings e and d can be composed in order to translate directly from state transitions to displays:

$$\begin{array}{lcl} \gamma \circ \mu & = & (\backslash_{d,\otimes,\delta}) \circ (\backslash_{e,\oplus,\delta}) \\ & = & \backslash_{d\circ e,\otimes,\delta} \end{array}$$

Giving the following mapping:

$$(\gamma \circ \mu)^* : [[E]] \rightarrow [D]$$

3.6 Key Types, Mappings and Filmstrip Laws

The filmstrip production pattern identifies several key definitions and some laws that the definitions must satisfy. The following key components must be provided: data types for events E, semantics S, and displays D. The semantics and displays should be monoids, and the semantics may include further domain-specific constraints. The events should provide an identity mapping i that is the basis for a standard state-transition mapping σ that must satisfy the specification in section 3.3. The mapping from states to displays can be constructed from two mappings e and d from events to semantic values and from semantic values to displays respectively. The semantic value mapping and the display monoid must satisfy the equations defined in section 3.5. The next section uses the language ESL to implement the pattern.

4 PATTERN REPRESENTATION IN ESL

The previous section has defined a filmstrip pattern that is independent of any implementation language and the data types used for events, semantics, and the displays. This section uses polymorphic functions in ESL to define the pattern in terms of its constituent mappings: section 4.1 defines the state transition mapping and then section 4.2 defines ω that expects the pattern component mappings as arguments.

4.1 State Transitions

State transitions are implemented using a structure $([E], i : E \rightarrow \text{Int})$ and a mapping σ that maps a history of events to a sequence of state transitions where each state is a sub-sequence of the events. The ESL definition of σ is shown in figure 7. The function `combine` is used to ensure that the specification of σ, as defined in section 3.3, is satisfied.

For example if E = `Message` then given the following sequence of messages:

```
h = [PreyAt(10,20),
     PredAt(1,20,10),
     PreyAt(9,20),
     PredAt(1,19,11),
     ...]
```

`σ[Message](h)` produces:

```
combine[E](i::(E) → Int,ids::[Int],h::[E],m::E)::[E] =
  case h {
    [] → if member[Int](i(m),ids) then [] else [m];
    hh::[T] + [mm::E] →
      if member[Int](i(m),ids)
      then combine[E](i,ids,hh,mm);
      else combine[E](i,ids+[i(m)],hh,mm) + [m];
  }

σ[E](i::(E) → Int,h::[E])::[[E]] =
  case h {
    []            → [];
    hh::[E]+[m::E] → σ[T](i,hh) + [combine[E](i,[],hh,m)];
  }
```

Figure 7: State Transitions in ESL

```
map[M,N](f::(M) → N,l::[M])::[N] =
  case l {
    m::M;
    ms::[M];
    [][M] → [][N];
    m:ms → (f(m)):map[M,N](f,ms);
  }

foldr[M,N](map::(M) → N,op::(N,N) → N,empty::N,list::[M])::N =
  case list {
    [] → empty;
    h::M:t::[M] → op(map(h),foldr[M,N](map,op,empty,t));
  }

ω[E,S,D](events::[E],
         i::(E) → Int,
         e::(E) → S,
         d::(S) → D,
         ⊗::(D,D) → D,
         δ::D)::[D] =
  let m::(ms::[E])::D = foldr[E,D](d∘e,⊗,δ,ms)
  in map[[E],D](m,σ[E](i,events))
```

Figure 8: Filmstrip Pattern Implemented in ESL

```
[[],
 [PreyAt(10,20)],
 [PreyAt(10,20),PredAt(1,20,10)],
 [PredAt(1,20,10),PreyAt(9,20)],
 [PreyAt(9,20),PredAt(1,19,11)],
 ...]
```

Theorem 4.1. *The definition of σ given above satisfies the requirements 1, 2 and 3 given in section 3.3.*

Proof: By induction on the length of h.

4.2 Filmstrip Mapping in ESL

The filmstrip mapping ω maps sequences of events to sequences of displays. It relies on constituent mappings as defined in figure 6 and combines them using `foldr` (that implements _,_,_) and `map` (that implements _*). Figure 8 shows the definition of the mapping in ESL.

5 PREDATOR-PREY FILMSTRIPS IN ESL

Given a collection of messages generated by ESL actors, the filmstrip is created as an ESL sequence of display elements by supplying ω with the messages, mappings and display monoid components:

```
filmstrip(messages::[Message])::[Tree] = ω[Message,Board,Tree]
  (messages,id,mapMessage,mapBoard,mergeDisplays,emptyDisplay)
```

The event history data type `Messages` has already been defined and the definition of `id` is:

```
id(PredAt(id::Int,_,_))::Int  = id;
id(_)::Int                    = -1;
```

This section provides the ESL definitions of: the semantic domain `Board` in section 5.1; the semantic mapping `mapMessage` in section 5.2; displays `Tree` and `mergeDisplays` in section 5.3; and, the display mapping `mapBoard` in 5.4 which also includes an example translation from a sequence of predator-prey messages to the resulting filmstrip.

5.1 Semantic Domain

The semantic domain $(S, \oplus, \epsilon)$ is used to represent a whole-system representation of a state. The predator-prey semantic domain is a *board* that contains locations. Each location can be *empty*, a *rock*, a *predator* or a *prey*. The location elements can be represented as a single data type and the board is a two-dimensional list:

```
data Location = EmptyLoc | PredLoc | PreyLoc | Rock;
type Board    = [[Location]];
```

The semantic domain should form a monoid. The following is the empty board:

```
emptyBoard::Board =
 [[if member[Pos](Point(x,y),rocks)
   then Rock
   else EmptyLoc
  | x::Int ← 0..width]
 | y::Int ← 0..height];
```

The monoid combination operation must be associative and have `emptyBoard` as a left and right identity assuming rocks are always in the same place:

```
mergeBoards(b1::Board,b2::Board)::Board = [[
  mergeLocs(b1[x][y],b2[x][y] | x::Int ← width] | y::Int ← height];

mergeLocs(Rock,l::Location)::Location      = Rock;
mergeLocs(l::Location,Rock)::Location      = Rock;
mergeLocs(l::Location,EmptyLoc)::Location  = l;
mergeLocs(EmptyLoc,l::Location)::Location  = l;
mergeLocs(PredLoc,PredLoc)::Location       = PredLoc;
mergeLocs(PreyLoc,PreyLoc)::Location       = PreyLoc;
```

The semantic domain structure for the predator-prey application is therefore `(Board,mergeLocs,emptyBoard)`.

5.2 Semantic Mapping

The semantic mapping must translate a state into a semantic value. Since the source and target of the semantic mapping both form a monoid, the mapping can be generated using a map for a single event as follows:

```
mapMessage(m::Message)::Board =
  case m {
    PredAt(_,x0::Int,y0::Int) →
     [[ if (x=x0) and (y=y0)
        then PredLoc
        else
         if member[Pos](Point(x,y),rocks)
         then Rock
         else EmptyLoc
      | x::Int ← 0..width]
     | y::Int ← 0..height];
    PreyAt(x0::Int,y0::Int) →
      [[ if (x=x0) and (y=y0)
         then PreyLoc
         else
          if member[Pos](Point(x,y),rocks)
          then Rock
          else EmptyLoc
        | x::Int ← 0..width]
       | y::Int ← 0..height]
  }
```

The semantic mapping can be defined as follows:

```
μ = foldr[Message,Board](mapMessage,mergeBoards,emptyBoard)
```

and generalised to μ^* using the definition of `map` as required. The history is therefore produced by copying forward snapshot fragments until an actor causes a change when it processes a message.

5.3 Displays

The filmstrips are represented as sequences of displays. ESL provides a number of display types that can be used to populate the filmstrip pattern. The predator-prey example can be displayed as a two-dimensional board that can be represented as a nested collection of trees containing horizontal and vertical boxes, shapes and images:

```
data Tree =
  TreeNode([Shape])   // A picture made up of shapes.
| VBox([Tree])        // A box of elements arranged vertically.
| HBox([Tree])        // A box of elements arranged horizontally.

data Shape
  Rectangle(Int,Int)  // Rectangle(width,height).
| Circle(Int)         // Circle(radius).
| Line(Int)           // Line(length).
| Image(Int,Int,Str)  // Image(width,height,location).
| Text(Str);          // Text(string).
```

A space can be represented using:

```
space::Tree    = TreeNode(Rectangle(size,size));
```

A simple two-dimensional board representing a predator-prey display can be represented as follows:

```
VBox([
  HBox([space,Image(size,size,'rock.png')]),
  HBox([Image(size,size,'wolf.png'),Image(size,size,'sheep.png')])
])
```

To comply with the filmstrip pattern, the language of displays must form a monoid. In the case of a two-dimensional predator-prey world the empty display is:

```
VBox([
  HBox([space,space]),
  HBox([space,space])
])
```

The binary display combination operator is implemented as follows where `l[i]` indexes an element in a list:

```
mergeDisplays(d1::Tree,d2::Tree)::Tree =
  case d1,d2 {
    VBox(l1::[Tree]),VBox(l2::[Tree]) →
      VBox([ mergeDisplays(l1[i],l2[i]) | i::Int ← 0..|l1|)]);
    HBox(l1::[Tree]),HBox(l2::[Tree]) →
      HBox([ mergeDisplays(l1[i],l2[i]) | i::Int ← 0..|l1|]);
    _,_ when d1 = space → d2;
    _,_ when d2 = space → d1;
    _,_ when d1 = d2    → d1;
  }
```

Assuming that `emptyDisplay` is a tree of the appropriate size and shape that contains only spaces then the displays form a monoid `(Tree,mergeDisplays,emptyDisplay)` as required.

```
[VBox([
   HBox([TreeNode(Rectangle(30,30)),
         TreeNode(Image(30,30,'sheep.jpg')),
         TreeNode(Rectangle(30,30))]),
   HBox([TreeNode(Rectangle(30,30)),
         TreeNode(Rectangle(30,30)),
         TreeNode(Rectangle(30,30))]),
   HBox([TreeNode(Rectangle(30,30)),
         TreeNode(Rectangle(30,30)),
         Image(30,30,'rock.png')))])])),
 VBox([
   HBox([TreeNode(Rectangle(30,30)),
         TreeNode(Image(30,30,'sheep.jpg')),
         TreeNode(Rectangle(30,30))]),
   HBox([TreeNode(Image(30,30,'wolf.jpg')),
         TreeNode(Rectangle(30,30)),
         TreeNode(Rectangle(30,30))]),
   HBox([TreeNode(Rectangle(30,30)),
         TreeNode(Rectangle(30,30)),
         Image(30,30,'rock.png')))])])),
 VBox([
   HBox([TreeNode(Rectangle(30,30)),
         TreeNode(Rectangle(30,30)),
         TreeNode(Rectangle(30,30))]),
   HBox([TreeNode(Image(30,30,'wolf.jpg')),
         TreeNode(Image(30,30,'sheep.jpg')),
         TreeNode(Rectangle(30,30))]),
   HBox([TreeNode(Rectangle(30,30)),
         TreeNode(Rectangle(30,30)),
         Image(30,30,'rock.png')))])])),
 VBox([
    HBox([TreeNode(Rectangle(30,30)),
          TreeNode(Rectangle(30,30)),
          TreeNode(Rectangle(30,30))]),
    HBox([TreeNode(Rectangle(30,30)),
          TreeNode(Image(30,30,'sheep.jpg')),
          TreeNode(Rectangle(30,30))]),
    HBox([TreeNode(Image(30,30,'wolf.jpg')),
          TreeNode(Rectangle(30,30)),
          Image(30,30,'rock.png')))])]))]
```

Figure 9: A Filmstrip

5.4 Display Mapping

Since the semantic domain and displays both form monoids, the display mapping can be generated from a single map from a board to a tree:

```
rockIcon::Tree     = TreeNode(Image(size,size,'rock.png'));
predIcon::Tree     = TreeNode(Image(size,size,'wolf.jpg'));
preyIcon::Tree     = TreeNode(Image(size,size,'sheep.jpg'));
emptyDisplay::Tree = mapBoard(emptyBoard);

mapBoard(b::Board)::Tree =
  let mapRow(row::[Location])::Tree =
        HBox([ case l {
                 PredLoc  → predIcon;
                 PreyLoc  → preyIcon;
                 EmptyLoc → space;
                 Rock     → rockIcon
               } | l::Location ← row ])
  in VBox([ mapRow(b[y]) | y::Int ← 0..height ]);
```

Consider a 3-by-3 predator-prey world with a single rock at position (2, 2). Given the following messages:

```
messages = [PreyAt(1,0),PredAt(1,0,1),PreyAt(1,1),PredAt(1,0,2)]
```

the filmstrip produced by `filmstrip(messages)` is shown in figure 9.

6 IMPLEMENTATION

ESL source code is translated to an instruction set that is executed on a virtual machine written in Java. Graphics libraries are integrated with ESL both in terms of the type system and the run-time in order to support a range of different displays that can be used to show the results of actor-based programs. Figure 10 shows example filmstrips that have been produced by ESL applications.

Figure 10a shows a snapshot of an ESL filmstrip that simulates a shop consisting of assistants, customers and criminals. The customers wait for help while browsing and queue at tills to be served. Figure 10b shows the event and semantic domains for the shop: the customer, assistant and till ids are defined to be integers and a shop state is `Shop(o,a,b,h,t)` where `o` are the customers outside the shop, `a` are the assistants on the floor, `b` are the customers who are browsing, `h` represents customers waiting for help and being helped, and `t` are the tills that may have an assistant serving and have a (possibly empty) queue of customers.

Figure 10c shows a snapshot of an ESL filmstrip that simulates traffic flow at a junction. The junction state is shown in figure 10d where `Road(left_l,right_l,left_r,j,right_r)` contains the current state of the left and right traffic lights (`left_l` and `right_l`) the traffic flow queuing and leaving the left and right roads (`left_r` and `right_r`) and the car passing the single-track mid-point of the junction `j`.

Figure 10e shows a snapshot of an ESL filmstrip that implements the dining philosophers. The state is shown in figure 10f where `Dining(f,p)` contains the fork identifiers `f` and the philosopher states `p` where a philosopher state is `Phil(i,l,r)` where `i` is the identifier of the philosopher and `l` and `r` are the left and right forks or `Nothing` if the philosopher is not holding a fork.

In all cases it is possible to map the events defined in figures 10b, 10d and 10f to their respective semantic data domains. Furthermore, it should be clear that the semantic domains form monoids based on an empty value and a binary composition operator.

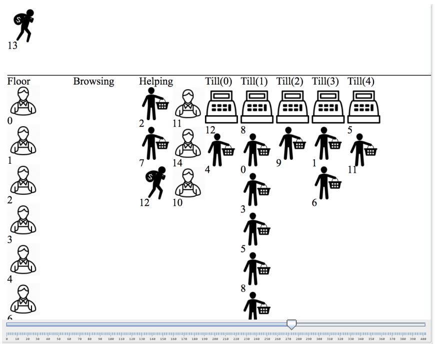
7 RELATED WORK

The general notion of sense-making processes originated in the field of intelligence analysis through the seminal work of Pirolli and Card [32]. However, there is a growing amount of work that addresses the problems of the more specialised form of sense-making, *debugging* of actor and agent-based systems such as [39]. The latter for example, exhibits many of the characterisation of the Pirolli-Card sense-making process such as re-representation through multiple complementary abstractions of the underlying multi-agent system to identify new insights and actions.

Many of the implementation level concerns of sense-making and debugging are described in [25]. The tool in [28] produces static diagrams of agent communication topologies using a *society tool*. They support off-line video-style replay facilities with forward and backward video modes as a powerful sense-making aid, although the structure of agents seems to be fixed.

The challenges and an approach to source-level agent-based debugging is described in [22]. Several systems support run-time instrumentation of actor-based systems, for example [36] and [23] which monitors a run-time system for semantic properties. Our work differs from all these approaches since we aim to understand a system in terms of its solution-domain instead of the implementation-domain.

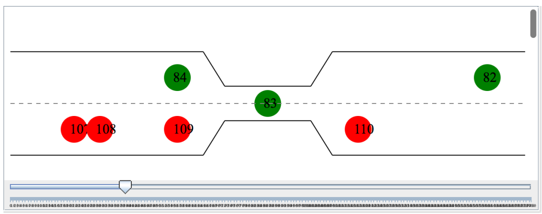
The use of system traces to address the challenge of understanding the behaviour of MAS is described by Búrdalo *et al.* in [6] where a standard model of trace data and an architecture that supports trace-processing is presented. The work in [6] does not describe



(a) Shop Filmstrip

```
type Cid          = Int;
type Aid          = Int;
type Tid          = Int;
data ShopE        = NotInShop(Cid)
                  | Browsing(Cid)
                  | Queueing(Cid,Tid)
                  | SeekingHelp(Cid)
                  | GettingHelp(Cid,Aid)
                  | OnFloor(Aid)
                  | AtTill(Aid,Tid);
data Helping      = Help(Cid,Possibly[AId]);
data Possibly[T]  = Just(T) | Nothing;
data Till         = Till(Tid,Possibly[Aid],[Cid]);
data ShopS        = Shop([Cid],[Aid],[Cid],[Helping],[Till]);
```

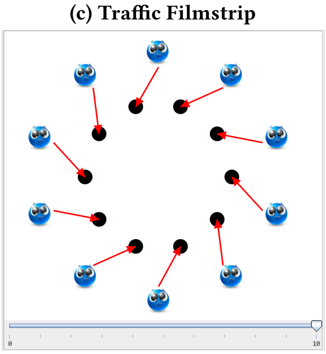
(b) Shop Event and Semantic Domains



(c) Traffic Filmstrip

```
type Vid           = Int;
data TrafficLight  = Left | Right;
data Colour        = Red | Amber | Green;
data RoadE         = QueueLeft(Vid)
                   | QueueRight(Vid)
                   | Advance(Vid)
                   | LeaveLeft(Vid)
                   | LeaveRight(Vid)
                   | Change(TrafficLight,Color);
data Road          = Road([Vid],[Vid]);
data RoadS = Road(Colour,Colour,Road,Possibly[Vid],Road);
```

(d) Traffic Event and Semantic Domains



(e) Dining Philosophers Filmstrip

```
type Pid           = Int;
type Fid           = Int;
type Forks         = [Fid];
data Side          = Left | Right;
data Philosopher   = Phil(Pid,Possibly[Fid],Possibly[Fid]);
type Philosophers  = [Philosopher]
data DinerE        = Pickup(Philosopher,Fid,Side)
                   | Eat(Pid)
                   | DropFork(Philosopher,Fid,Side);
data DinerS        = Dining(Forks,Philosophers);
```

(f) Dining Philosopher Event and Semantic Domains

Figure 10: Filmstrip Examples

how to process the trace data and the pattern presented in this paper could be incorporated into that work.

Other approaches to sense-making include the interrogation of system traces and source-level debuggers. The visualization of Java execution traces in terms of object diagrams and sequence diagrams is proposed as a means for sense-making in [20]. Our approach provides a structured framework for defining many types of diagram including object and sequence. Queries are applied to AgentSpeak execution histories [41] in order to determine whether certain behaviours occurred. The ESL language supports similar queries (described in [10]) which are complementary to the animations described in this paper.

Model-checking can be used to formally express system properties of actor-based systems, for example [19] uses a model-checker called McErlang to check safety properties of Timed Rebeca that are translated to Erlang. Whilst this approach can be very successful for particular types of properties, we argue that the approach described in this paper has wider application and is more scalable.

Process event logs can be used as a basis for analysis of complex business applications. The event logs are similar to the histories described in this paper. In some cases visualisation has been used to compare different processes [5, 42], although there is no description of a general pattern for constructing the visual output.

Filmstrips, first attributed to D'Souza and Wills [12] provide important visual support for examining histories. In their approach, a filmstrip is a set of contiguous snapshots that describe how a system state evolves through a specific scenario. Filmstrips have also been applied in areas such as functional testing [7]. Efforts to incorporate filmstrips, include the recent efforts by Gogolla *et al.* use filmstrip models for automatic validation of model dynamics of applications [18]. Gil and Kent, in 1995, proposed the use of filmstrips as an important component for three dimensional software modelling in an effort to move away from a topological graph metaphor [17].

8 CONCLUSION

Actor-based systems exhibit non deterministic behaviour that makes sense-making activities such as debugging challenging. Semantically based visualisation is a powerful tool in helping understand such execution. We have described how *filmstrips* can be used to examine histories of executions and hence function as a sense making tool. We have presented a generalisation of the necessary machinery (event histories, domain-independent filmstrip representation and the operations possible over the event histories) as an algebraic pattern. The pattern has potential for use in environments where agent based simulation histories are key output for analysis.

The pattern has been implemented in the open-source language ESL that supports both actors and polymorphic functions, and has been used to implement a number of actor-based applications including predator-prey, shop and traffic simulations, and dining philosophers.

REFERENCES

[1] Yoosef Abushark, John Thangarajah, Tim Miller, and James Harland. Checking consistency of agent designs against interaction protocols for early-phase defect location. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, pages 933–940. International Foundation for Autonomous Agents and Multiagent Systems, 2014.

[2] Hadi Alizadeh Ara, Amir Behrouzian, Marc Geilen, Martijn Hendriks, Dip Goswami, and Twan Basten. Analysis and visualization of execution traces of dataflow applications. *IDEA 2016: Integrating Dataflow, Embedded Computing, and Architecture*, page 19, 2017.

[3] José Barbosa, Paulo Leitão, Emmanuel Adam, and Damien Trentesaux. Dynamic self-organization in holonic multi-agent manufacturing systems: The adacor evolution. *Computers in industry*, 66:99–111, 2015.

[4] Bernd Bruegge, Tim Gottschalk, and Bin Luo. A framework for dynamic program analyzers. In *ACM SIGplan Notices*, volume 28, pages 65–82. ACM, 1993.

[5] Joos CAM Buijs and Hajo A Reijers. Comparing business process variants using models and event logs. In *Enterprise, Business-Process and Information Systems Modeling*, pages 154–168. Springer, 2014.

[6] Luis Búrdalo, Andrés Terrasa, Vicente Julián, and Ana García-Fornes. Trammas: A tracing model for multiagent systems. *Engineering Applications of Artificial Intelligence*, 24(7):1110–1119, 2011.

[7] Tony Clark. Model based functional testing using pattern directed filmstrips. In *Automation of Software Test. AST'09. ICSE Workshop on*, pages 53–61. IEEE, 2009.

[8] Tony Clark, Vinay Kulkarni, Souvik Barat, and Balbir Barn. Actor monitors for adaptive behaviour. In *Proceedings of the 10th Innovations in Software Engineering Conference, ISEC 2017, Jaipur, India, February 5-7, 2017*, pages 85–95, 2017.

[9] Tony Clark, Vinay Kulkarni, Souvik Barat, and Balbir Barn. ESL: an actor-based platform for developing emergent behaviour organisation simulations. In *Advances in Practical Applications of Cyber-Physical Multi-Agent Systems: The PAAMS Collection - 15th International Conference, PAAMS 2017, Porto, Portugal, June 21-23, 2017, Proceedings*, pages 311–315, 2017.

[10] Tony Clark, Vinay Kulkarni, Balbir Barn, and Souvik Barat. The construction and interrogation of actor based simulation histories. In *Proceedings of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th International Conference on Conceptual Modelling, Valencia, Spain, 2017.*, 2017.

[11] Célian Colon, David Claessen, and Michael Ghil. Bifurcation analysis of an agent-based model for predator–prey interactions. *Ecological Modelling*, 317: 93–106, 2015.

[12] Desmond D'Souza and Alan Wills. Catalysis. practical rigor and refinement: Extending omt, fusion, and objectory, 1995. URL http://catalysis.org/publications/papers/1995-catalysis-fusion.pdf.

[13] Nuno Fachada, Vitor V Lopes, Rui C Martins, and Agostinho C Rosa. Towards a standard model for research in agent-based modeling and simulation. *PeerJ Computer Science*, 1:e36, 2015.

[14] Amro M Farid. Multi-agent system design principles for resilient coordination & control of future power systems. *Intelligent Industrial Systems*, 1(3):255–269, 2015.

[15] Jacques Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Multi Agent Systems, 1998. Proceedings. International Conference on*, pages 128–135. IEEE, 1998.

[16] Paul A Fishwick. Computer simulation: growth through extension. *Transactions of the Society for Computer Simulation*, 14(1):13–24, 1997.

[17] Joseph Gil and Stuart Kent. Three dimensional software modelling. In *Proceedings of the 20th international conference on Software engineering*, pages 105–114. IEEE Computer Society, 1998.

[18] Martin Gogolla, Lars Hamann, Frank Hilken, Mirco Kuhlmann, and Robert B France. From application models to filmstrip models: An approach to automatic validation of model dynamics. In *Modellierung*, volume 225, pages 273–288, 2014.

[19] Ali Jafari, Ehsan Khamespanah, Haukur Kristinsson, Marjan Sirjani, and Brynjar Magnusson. Statistical model checking of timed rebeca models. *Computer Languages, Systems & Structures*, 45:53–79, 2016.

[20] S Jayaraman, Bharat Jayaraman, and Demian Lessa. Compact visualization of java program execution. *Software: Practice and Experience*, 47(2):163–191, 2017.

[21] Rajesh K Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM, 2009.

[22] Vincent J Koeman, Koen V Hindriks, and Catholijn M Jonker. Designing a source-level debugger for cognitive agent programs. *Autonomous Agents and Multi-Agent Systems*, 31(5):941–970, 2017.

[23] Paul Lavery and Takuo Watanabe. An actor-based runtime monitoring system for web and desktop applications. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2017 18th IEEE/ACIS International Conference on*, pages 385–390. IEEE, 2017.

[24] Thomas J LeBlanc, John M Mellor-Crummey, and Robert J Fowler. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9(2):203–217, 1990.

[25] Carmen Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and Elisa Gonzalez Boix. A study of concurrency bugs and advanced development support for actor-based programs. *arXiv preprint arXiv:1706.07372*, 2017.

[26] Tom McDermott, William Rouse, Seymour Goodman, and Margaret Loper. Multi-level modeling of complex socio-technical systems. *Procedia Computer Science*, 16:1132–1141, 2013.

[27] Geoffrey P Morgan and Kathleen M Carley. An agent-based framework for active multi-level modeling of organizations. In *International Conference on Social Computing, Behavioral-Cultural Modeling and Prediction and Behavior Representation in Modeling and Simulation*, pages 272–281. Springer, 2016.

[28] Divine T. Ndumu, Hyacinth S. Nwana, Lyndon C. Lee, and Jaron C. Collis. Visualising and debugging distributed multi-agent systems. In *Proceedings of the Third Annual Conference on Autonomous Agents*, AGENTS '99, pages 326–333, New York, NY, USA, 1999. ACM. ISBN 1-58113-066-X.

[29] Brian Nielsen and Gul Agha. Semantics for an actor-based real-time language. In *Parallel and Distributed Real-Time Systems, 1996. Proceedings of the 4th International Workshop on*, pages 223–228. IEEE, 1996.

[30] Taleb AS Obaid. The predator-prey model simulation. *Basrah Journal of Science*, 31(2):103–109, 2013.

[31] Eloi Pereira, Christoph M Kirsch, Raja Sengupta, and João Borges de Sousa. Bigactors—a model for structure-aware computation. In *Cyber-Physical Systems (ICCPS), 2013 ACM/IEEE International Conference on*, pages 199–208. IEEE, 2013.

[32] Peter Pirolli and Stuart Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of international conference on intelligence analysis*, volume 5, pages 2–4, 2005.

[33] David Poutakidis, Lin Padgham, and Michael Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, pages 960–967. ACM, 2002.

[34] David Poutakidis, Lin Padgham, and Michael Winikoff. An exploration of bugs and debugging in multi-agent systems. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 1100–1101. ACM, 2003.

[35] David V Pynadath and Milind Tambe. An automated teamwork infrastructure for heterogeneous software agents and humans. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):71–100, 2003.

[36] Andrea Rosà, Lydia Y. Chen, and Walter Binder. Profiling actor utilization and communication in akka. In *Proceedings of the 15th International Workshop on Erlang*, Erlang 2016, pages 24–32, New York, NY, USA, 2016. ACM.

[37] Zhen Z Shi, Chih-Hang Wu, and David Ben-Arieh. Agent-based model: a surging tool to simulate infectious diseases in the immune system. *Open Journal of Modelling and Simulation*, 2(01):12, 2014.

[38] Cameron Taylor, Heather Koyuk, Jessica Coyle, Russell Waggoner, and Kelly Newman. An agent-based model of predator-prey relationships between transient killer whales and other marine mammals. 2007.

[39] Marc H Van Liedekerke and Nicholas M Avouris. Debugging multi-agent systems. *Information and Software Technology*, 37(2):103–112, 1995.

[40] Karl E Weick. *Sensemaking in organizations*, volume 3. Sage, 1995.

[41] Michael Winikoff. Debugging agent programs with why?: Questions. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 251–259. International Foundation for Autonomous Agents and Multiagent Systems, 2017.

[42] Sen Yang, Xin Dong, Moliang Zhou, Xinyu Li, Shuhong Chen, Rachel Webman, Aleksandra Sarcevic, Ivan Marsic, and Randall S Burd. Vit-pla: Visual interactive tool for process log analysis. In *KDD Workshop on Interactive Data Exploration and Analytics*, 2016.